FREE eBook

LEARNING

tcl

Free unaffiliated eBook created from
Stack Overflow contributors.

#tcl

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: tcl

It is an unofficial and free tcl ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official tcl.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with tcl

Remarks

Tcl is a cross platform language with full unicode support.

Flexibility: redefine or enhance existing commands or write new commands.

Event driven programming: Event driven I/O and variable tracing.

Library Interface: It is very easy to integrate existing C libraries into Tcl and provide a Tcl interface to the C library. These interface "stubs" are not tied to any particular version of Tcl and will continue to work even after upgrading Tcl.

Tcl Interface: Tcl provides a complete API so you use the Tcl interpreter from within your C/Python/Ruby/Java/R program.

Versions

Version	Notes	Release Date
8.6.6	Current Patch Release.	2016-07-27
8.6.5		2016-02-29
8.6.4		2015-03-12
8.6.3		2014-11-12
8.6.2		2014-08-27
8.6.1		2013-09-20
8.6.0	Current recommended version series for new code. Introduced object system and non-recursive execution engine.	2013-09-20
8.5.19	Current LTS release	2016-02-12
8.5.18		2015-03-

Version	Notes	Release Date
		06
8.5.17		2014-10-25
8.5.16		2014-08-25
8.5.15		2013-09-18
8.5.14		2013-04-03
8.5.13		2012-11-12
8.5.12		2012-07-27
8.5.11		2011-11-04
8.5.10		2011-06-24
8.5.9		2010-09-08
8.5.8		2009-11-16
8.5.7		2009-04-15
8.5.6		2008-12-23
8.5.5		2008-10-15
8.5.4		2008-08-15
8.5.3		2008-06-30

Version	Notes	Release Date
8.5.2		2008-03-28
8.5.1		2008-02-05
8.5.0	Current oldest supported version. Introduced expansion syntax, dictionaries and ensemble commands.	2007-12-20
8.4.20	Final 8.4 series release. *There will be no further releases of 8.4.*	2013-06-01
8.4.19		2008-04-18
8.4.18		2008-02-08
8.4.17		2008-01-04
8.4.16		2007-09-21
8.4.15		2007-05-25
8.4.14		2006-10-19
8.4.13		2006-04-19
8.4.12		2005-12-03
8.4.11		2005-06-28
8.4.10		2005-06-04
8.4.9		2004-12-07
8.4.8		2004-11-22

Version	Notes	Release Date
8.4.7		2004-07-25
8.4.6		2004-03-01
8.4.5		2003-11-24
8.4.4		2003-07-22
8.4.3		2003-05-19
8.4.2		2003-03-03
8.4.1		2002-10-22
8.4.0	First release by Tcl Core Team. Many performance enhancements. Improved 64-bit support.	2002-09-18
8.3.5		2002-10-18
8.3.4		2001-10-19
8.3.3		2001-04-06
8.3.2		2000-08-09
8.3.1		2000-04-26
8.3.0	Performance improvements.	2000-02-10
8.2	Stabilisation release	1999-08-18
8.1	Introduced Unicode support.	1999-04-30

Version	Notes	Release Date
8.0	Introduced bytecode compilation engine	1997-08-16

Examples

Installation

Installing **Tcl 8.6.4** on **Windows** :

1. The easiest way to get Tcl on a windows machine is to install the **ActiveTcl** distribution from ActiveState.
2. Navigate to www.activestate.com and follow the links to download the Free Community Edition of ActiveTcl for Windows (choose 32/64 bit version appropriately).
3. Run the installer which will result in a fresh install of ActiveTcl usually in the **C:\Tcl** directory.
4. Open a command prompt to test the install, type in "tclsh" which should open an interactive tcl console. Enter "info patchlevel" to check the version of tcl that was installed and it should display an output of the form "8.6.x" depending on the edition of ActiveTcl that has been downloaded.

- You may also want to add "C:\Tcl\bin" or its equivalent to your environment **PATH** variable.

```
C:\>tclsh
% info patchlevel
8.6.4
```

The Hello, world program in Tcl (and Tk)

The following code can be entered in a Tcl shell (`tclsh`), or into a script file and run through a Tcl shell:

```
puts "Hello, world!"
```

It gives the string argument `Hello, world!` to the command `puts`. The `puts` command writes its argument to standard out (your terminal in interactive mode) and adds a newline afterwards.

In a Tk-enabled shell, this variation can be used:

```
pack [button .b -text "Hello, world!" -command exit]
```

It creates a graphic button with the text `Hello, world!` and adds it to the application window. When pressed, the application exits.

A Tk-enabled shell is started as: `wish` Or using `tclsh` along with the following statement:

```
package require Tk
```

Features of Tcl

- Cross Platform Portability
  - Runs on Windows, Mac OS X, Linux, and virtually every variant of unix.
- Event driven programming
  - Trigger events based on variable read / write / unset.
  - Trigger events when a command is entered or left.
  - Trigger events when a I/O channel (file or network) becomes readable / writable.
  - Create your own events.
  - Trigger a command based on a timer.
- Object Oriented Programming
  - Mixins.
  - Superclasses and subclasses.
- Simple Grammar
- Full unicode support
  - It just works. No special commands are needed to handle unicode strings.
  - Convert to and from different encoding systems with ease.
- Flexible
  - Create new control structures and commands.
  - Access variables in the calling procedure's context.
  - Execute code in the calling procedure's context.
- Powerful introspection capabilities.
  - Many Tcl debuggers have been written in Tcl.
- Library interface
  - Integrate existing C libraries and provide a Tcl interface to the library.
  - Library "stubs" are not tied to any particular version of Tcl and will still work after a Tcl upgrade.
- Complete API
  - Embed a Tcl interpreter into your favorite language.
  - Python, Ruby, R, Java and others include a Tcl API.
- Embedded bigint library.
  - No special actions are needed to handle very large numerics.
- Safe interpreters
  - Create sandboxes in which user code can be run.
  - Enable and disable specific commands for the interpreter.
- Regular Expressions
  - A powerful and fast regular expression engine written by Henry Spencer (creator of regex).

Installing packages through teacup

Now days many languages are supporting **archive** server to install their packages into your local

machine. TCL also having same archive server we called it as Teacup

```
teacup version
teacup search <packageName>
```

Example

```
teacup install Expect
```

Read Getting started with tcl online: https://riptutorial.com/tcl/topic/3029/getting-started-with-tcl

Chapter 2: Control Structures

Syntax

- if expr1 ?then? body1 elseif expr2 ?then? body2 ... ?else? ?bodyN?
- for start test next body
- while test body
- foreach varlist1 list1 ?varlist2 list2 ...? body

Remarks

Documentation: break, for, foreach, if, switch, uplevel, while

Examples

Adding a new control structure to Tcl

In Tcl, a control structure is basically just another command. This is one possible implementation of a `do ... while` / `do ... until` control structure.

```
proc do {body keyword expression} {
    uplevel 1 $body
    switch $keyword {
        while {uplevel 1 [list while $expression $body]}
        until {uplevel 1 [list while !($expression) $body]}
        default {
            return -code error "unknown keyword \"$keyword\": must be until or while"
        }
    }
}
```

Common for both kinds of `do`-loops is that the script named `body` will always be executed at least once, so we do that right away. The invocation `uplevel 1 $body` means "execute script at the caller's stack level". This way, all variables used by the script will be visible, and any results produced will stay at the caller's level. The script then selects, based on the `keyword` parameter, whether to iterate while a condition is true or until it is false, which is the same as iterating while the logical negation of the condition is true. If an unexpected keyword is given, an error message is produced.

if / while / for

if *expr1* ?then? *body1* elseif *expr2* ?then? *body2* ... ?else? *?bodyN?*

exprN is an expression that evaluates to a boolean value. *bodyN* is a list of commands.

```
set i 5
```

```
if {$i < 10} {
  puts {hello world}
} elseif {$i < 70} {
  puts {enjoy world}
} else {
  puts {goodbye world}
}
```

for *start test next body*

start, *next* and *body* are lists of commands. *test* is an expression that evaluates to a boolean values.

The **break** command will break out of the loop. The **continue** command will skip to the next iteration of the loop.

The common usage is:

```
for {set i 0} {$i < 5} {incr i} {
  puts "$i: hello world"
}
```

Since *start* and *next* are lists of commands, any command may be present.

```
for {set i 0; set j 5} {$i < 5} {incr i; incr j -1} {
  puts "i:$i j:$j"
}
```

while *test body*

The *test* is any expression that evaluates to a boolean value. While *test* is true, *body* is executed.

```
set x 0
while {$x < 5} {
  puts "hello world"
  incr x
}
```

The **break** command will break out of the loop. The **continue** command will skip to the next iteration of the loop.

```
set lineCount 0
while {[gets stdin line] >= 0} {
  puts "[incr lineCount]: $line"
  if { $line eq "exit" } {
    break
  }
}
```

List iteration: foreach

foreach *varlist1 list1 ?varlist2 list2 ...? body*

foreach is a powerful control structure that allows looping over a list or multiple lists.

```
set alpha [list a b c d e f]
foreach {key} $alpha {
   puts "key: $key"
}
```

Multiple variable names may be specified.

```
set alphaindexes [list a 1 b 2 c 3 d 4 e 5 f 6]
foreach {key num} $alphaindexes {
   puts "key:$key num:$num"
}
```

Multiple lists can be iterated over at the same time.

```
set alpha [list a b c d e f]
set indexes [list 1 2 3 4 5 6]
foreach {key} $alpha {idx} $indexes {
   puts "key: $key idx:$idx"
}
```

Read Control Structures online: https://riptutorial.com/tcl/topic/4723/control-structures

Chapter 3: Dictionaries

Remarks

Dictionaries in Tcl are *values* that hold a mapping from arbitrary values to other arbitrary values. They were introduced in Tcl 8.5, though there are limited versions for (the now unsupported) Tcl 8.4. Dictionaries are syntactically the same as lists with even numbers of elements; the first pair of elements is the first key and value of the dictionary, the second pair is the second tuple.

Thus:

```
fox "quick brown" dogs "lazy"
```

is a valid dictionary. The same key can be multiple times, but it is exactly as if the latter's value was in the earlier's value; these are the same dictionary:

```
abcd {1 2 3} defg {2 3 4} abcd {3 4 5}
```

```
abcd {3 4 5} defg {2 3 4}
```

Whitespace is unimportant, just as with lists.

An important concept with dictionaries is iteration order; dictionaries try to use the key insertion order as their iteration order, though when you update the value for a key that already exists, you overwrite that key's value. New keys go on the end.

References: dict

Examples

List-appending to a nested dictionary

If we have this dictionary:

```
set alpha {alice {items {}} bob {items {}} claudia {items {}} derek {items {}}}
```

And want to add "fork" and "peanut" to Alice's items, this code won't work:

```
dict lappend alpha alice items fork peanut
dict get $alpha alice
# => items {} items fork peanut
```

Because it would be impossible for the command to know where the key tokens end and the values to be list-appended start, the command is limited to one key token.

The correct way to append to the inner dictionary is this:

```
dict with alpha alice {
    lappend items fork peanut
}
dict get $alpha alice
# => items {fork peanut}
```

This works because the `dict with` command lets us traverse nested dictionaries, as many levels as the number of key tokens we provide. It then creates variables with the same names as the keys on that level (only one here: `items`). The variables are initialized to the value of the corresponding item in the dictionary. If we change the value, that changed value is used to update the value of the dictionary item when the script ends.

(Note that the variables continue to exist when the command has ended.)

Basic use of a dictionary

Creating a dictionary:

```
set mydict [dict create a 1 b 2 c 3 d 4]
dict get $mydict b ; # returns 2
set key c
set myval [dict get $mydict $key]
puts $myval
# remove a value
dict unset mydict b
# set a new value
dict set mydict e 5
```

Dictionary keys can be nested.

```
dict set mycars mustang color green
dict set mycars mustang horsepower 500
dict set mycars prius-c color orange
dict set mycars prius-c horsepower 99
set car [dict get $mycars mustang]
# $car is: color green horsepower 500
dict for {car cardetails} $mycars {
  puts $car
  dict for {key value} $cardetails {
    puts "  $key: $value"
  }
}
```

The dict get command can raise an error

```
set alpha {a 1 b 2 c 3}
dict get $alpha b
# => 2
dict get $alpha d
# (ERROR) key "d" not known in dictionary
```

If `dict get` is used to retrieve the value of a missing key, an error is raised. To prevent the error, use `dict exists`:

```
if {[dict exists $alpha $key]} {
    set result [dict get $alpha $key]
} else {
    # code to deal with missing key
}
```

How to deal with a missing key of course depends on the situation: one simple way is to set the `result` to a default "empty" value.

If the code never attempts to retrieve other keys that are in the dictionary, `dict get` will of course not fail. But for arbitrary keys, `dict get` is an operation that needs to be guarded. Preferably by testing with `dict exists`, though exception catching will work too.

Iterating over a dictionary

You can iterate over the contents of a dictionary with `dict for`, which is similar to `foreach`:

```
set theDict {abcd {ab cd} bcde {ef gh} cdef {ij kl}}
dict for {theKey theValue} $theDict {
    puts "$theKey -> $theValue"
}
```

This produces this output:

```
abcd -> ab cd
bcde -> ef gh
cdef -> ij kl
```

You'd get the same output by using `dict keys` to list the keys and iterating over that:

```
foreach theKey [dict keys $theDict] {
    set theValue [dict get $theDict $theKey]
    puts "$theKey -> $theValue"
}
```

But `dict for` is more efficient.

Read Dictionaries online: https://riptutorial.com/tcl/topic/4065/dictionaries

Chapter 4: Expressions

Remarks

Another benefit from using braced expression strings is that the byte compiler usually can generate more efficient code (5 - 10x faster) from them.

Examples

The problems with unbraced expressions

It is a good practice to provide expression string arguments as braced strings. The heading "Double Substitution" outlines important reasons behind the same.

The `expr` command evaluates an operator-based expression string to calculate a value. This string is constructed from the arguments in the invocation.

```
expr 1 + 2    ; # three arguments
expr "1 + 2"  ; # one argument
expr {1 + 2}  ; # one argument
```

These three invocations are equivalent and the expression string is the same.

The commands `if`, `for`, and `while` use the same evaluator code for their condition arguments:

```
if {$x > 0} ...
for ... {$x > 0} ... ...
while {$x > 0} ...
```

The main difference is that the condition expression string must always be a single argument.

As with every argument in a command invocation in Tcl, the contents may or may not be subjected to substitution, depending on how they are quoted / escaped:

```
set a 1
set b 2
expr $a + $b   ; # expression string is {1 + 2}
expr "$a + $b" ; # expression string is {1 + 2}
expr \$a + \$b ; # expression string is {$a + $b}
expr {$a + $b} ; # expression string is {$a + $b}
```

There is a difference in the third and fourth cases as the backslashes / braces prevent substitution. The result is still the same, since the evaluator inside `expr` can itself perform Tcl variable substitution and transform the string to `{1 + 2}`.

```
set a 1
set b "+ 2"
```

```
expr $a $b   ; # expression string is {1 + 2}
expr "$a $b" ; # expression string is {1 + 2}
expr {$a $b} ; # expression string is {$a $b}: FAIL!
```

Here we get into trouble with the braced argument: when the evaluator in `expr` performs substitutions, the expression string has already been parsed into operators and operands, so what the evaluator sees is a string consisting of two operands with no operator between them. (The error message is "`missing operator at _@_ in expression "$a _@_$b"`".)

In this case, variable substitution before `expr` was called prevented an error. Bracing the argument prevented variable substitution until expression evaluation, which caused an error.

Situations like this can occur, most typically when an expression to evaluate is passed in as a variable or parameter. In those cases there is no other choice than to leave the argument unbraced to allow the argument evaluator to "unpack" the expression string for delivery to `expr`.

In most other cases, though, bracing the expression does no harm and indeed can avert a lot of problems. Some examples of this:

Double substitution

```
set a {[exec make computer go boom]}
expr $a      ; # expression string is {[exec make computer go boom]}
expr {$a}    ; # expression string is {$a}
```

The unbraced form will perform the command substitution, which is a command that destroys the computer somehow (or encrypts or formats the hard disk, or what have you). The braced form will perform a variable substitution and then try (and fail) to make something of the string "[exec make computer go boom]". Disaster averted.

Endless loops

```
set i 10
while "$i > 0" {puts [incr i -1]}
```

This problem affects both `for` and `while`. While it seems that this loop would count down to 0 and exit, the condition argument to while is actually always `10>0` because that was what the argument was evaluated to be when the `while` command was activated. When the argument is braced, it is passed to the `while` command as `$i>0`, and the variable will be substituted once for every iteration. Use this instead:

```
while {$i > 0} {puts [incr i -1]}
```

Total evaluation

```
set a 1
if "$a == 0 && [incr a]" {puts abc}
```

What is the value of `a` after running this code? Since the `&&` operator only evaluates the right

operand if the left operand is true, the value should still be 1. But actually, it's 2. This is because the argument evaluator has already performed all variable and command substitutions by the time the expression string is evaluated. Use this instead:

```
if {$a == 0 && [incr a]} {puts abc}
```

Several operators (the logical connectives `||` and `&&`, and the conditional operator `?:`) are defined to *not* evaluate all their operands, but they can only work as designed if the expression string is braced.

Multiplying a variable by 17

```
set myVariable [expr { $myVariable * 17 }]
```

This shows how you can use a simple expression to update a variable. The `expr` command does not update the variable for you; you need to take its result and write it to the variable with `set`.

Note that newlines are not important in the little language understood by `expr`, and adding them can make longer expressions much easier to read.

```
set myVariable [expr {
    $myVariable * 17
}]
```

This does *exactly* the same thing though.

Calling a Tcl command from an expression

Sometimes you need to call a Tcl command from your expression. For example, supposing you need the length of a string in it. To do that, you just use a `[...]` sequence in the expression:

```
set halfTheStringLength [expr { [string length $theString] / 2 }]
```

You can call any Tcl command this way, but if you find yourself calling `expr` itself, ***stop!*** and think whether you really need that extra call. You can *usually* do just fine by putting the inner expression in parentheses.

Invalid bareword error

In Tcl itself, a string consisting of a single word does not need to be quoted. In the language of expression strings that `expr` evaluates, all operands must have an identifiable type.

Numeric operands are written without any decoration:

```
expr {455682 / 1.96e4}
```

So are boolean constants:

```
expr {true && !false}
```

Tcl variable substitution syntax is recognized: the operand will be set to the variable's value:

```
expr {2 * $alpha}
```

The same goes for command substitution:

```
expr {[llength $alpha] > 0}
```

Operands can also be mathematical function calls, with a comma-separated list of operands within parentheses:

```
expr {sin($alpha)}
```

An operand can be a double-quoted or braced string. A double-quoted string will be subject to substitution just like in a command line.

```
expr {"abc" < {def}}
```

If an operand isn't one of the above, it is illegal. Since there is no hint that shows what kind of a word it is, `expr` signals a bareword error.

Read Expressions online: https://riptutorial.com/tcl/topic/3052/expressions

Chapter 5: Pathnames and filenames

Syntax

- file dirname *filepath*
- file tail *filepath*
- file rootname *filepath*
- file extension *filepath*
- file join *path1 path2 ...*
- file normalize *path*
- file nativename *path*

Examples

Working with pathnames and filenames

```
% set mypath /home/tcluser/sources/tcl/myproject/test.tcl
/home/tcluser/sources/tcl/myproject/test.tcl
% set dir [file dirname $mypath]
/home/tcluser/sources/tcl/myproject
% set filename [file tail $mypath]
test.tcl
% set basefilename [file rootname $filename]
test
% set extension [file extension $filename]
.tcl
% set newpath [file join $dir .. .. otherproject]
/home/tcluser/sources/tcl/myproject/../../otherproject
% set newpath [file normalize $newpath]
/home/tcluser/source/otherproject
% set pathdisp [file nativename $newpath] ; # not on windows...
/home/tcluser/source/otherproject
% set pathdisp [file nativename C:$newpath] ; # on windows...
C:\home\tcluser\source\otherproject
% set normpath [file normalize $pathdisp]
C:/home/tcluser/source/otherproject
```

Documentation: file

Read Pathnames and filenames online: https://riptutorial.com/tcl/topic/5566/pathnames-and-filenames

Chapter 6: Procedure arguments

Remarks

References: proc
Argument Expansion (section 5)

Examples

A procedure that does not accept arguments

```
proc myproc {} {
    puts "hi"
}
myproc
# => hi
```

An empty argument list (the second argument after the procedure name, "myproc") means that the procedure will not accept arguments.

A procedure that accepts two arguments

```
proc myproc {alpha beta} {
    ...
    set foo $alpha
    set beta $bar     ;# note: possibly useless invocation
}

myproc 12 34          ;# alpha will be 12, beta will be 34
```

If the argument list consists of words, those will be the names of local variables in the procedure, and their initial values will be equal to the argument values on the command line. The arguments are passed by value and whatever happens to the variable values inside the procedure will not influence the state of data outside the procedure.

A procedure that accepts a variable number of arguments

```
### Definition
proc myproc {alpha {beta {}} {gamma green}} {
    puts [list $alpha $beta $gamma]
}
```

```
### Use
myproc A
# => A {} green
myproc A B
# => A B green
myproc A B C
```

```
# => A B C
```

This procedure accepts one, two, or three arguments: those parameters whose names are the first item in a two-item list are optional. The parameter variables (`alpha`, `beta`, `gamma`) get as many argument values as are available, assigned from left to right. Parameter variables that don't get any argument values instead get their values from the second item in the list they were a part of.

Note that optional arguments must come at the end of the argument list. If $argument_{N-1}$ is optional, $argument_N$ must be optional too. If in a case, where user have $argument_N$ but not $argument_{N-1}$, default value of $argument_{N-1}$ needs to be explicitly mentioned before $argument_N$, while calling the procedure.

```
myproc A B C D
# (ERROR) wrong # args: should be "myproc alpha ?beta? ?gamma?"
```

The procedure does not accept more than three arguments: note that a helpful error message describing the argument syntax is automatically created.

A procedure that accepts any number of arguments

```
proc myproc args { ... }
proc myproc {args} { ... } ;# equivalent
```

If the special parameter name `args` is the last item in the argument list, it receives a list of all arguments at that point in the command line. If there are none, the list is empty.

There can be arguments, including optional ones, before `args`:

```
proc myproc {alpha {beta {}} args} { ... }
```

This procedure will accept one or more arguments. The first two, if present, will be consumed by `alpha` and `beta`: the list of the rest of the arguments will be assigned to `args`.

A procedure that accepts a name/reference to a variable

```
proc myproc {varName alpha beta} {
    upvar 1 $varName var
    set var [expr {$var * $alpha + $beta}]
}
set foo 1
myproc foo 10 5
puts $foo
# => 15
```

In this particular case, the procedure is given the name of a variable in the current scope. Inside a Tcl procedure, such variables aren't automatically visible, but the `upvar` command can create an alias for a variable from another stack level: 1 means the caller's stack level, #0 means the global level, etc. In this case, the stack level 1 and the name `foo` (from the parameter variable `varName`) lets `upvar` find that variable and create an alias called `var`. Every read or write operation on `var` also

happens to `foo` in the caller's stack level.

The {*} syntax

Sometimes what you have is a list, but the command you want to pass the items in the list to demands to get each item as a separate argument. For instance: the `winfo children` command returns a list of windows, but the `destroy` command will only take a sequence of window name arguments.

```
set alpha [winfo children .]
# => .a .b .c
destroy $alpha
# (no response, no windows are destroyed)
```

The solution is to use the `{*}` syntax:

```
destroy {*}[winfo children .]
```

or

```
destroy {*}$alpha
```

What the `{*}` syntax does is to take the following value (no whitespace in between!) and splice the items in that value into the command line as if they were individual arguments.

If the following value is an empty list, nothing is spliced in:

```
puts [list a b {*}{} c d]
# => a b c d
```

If there are one or more items, they are inserted:

```
puts [list a b {*}{1 2 3} c d]
# => a b 1 2 3 c d
```

Read Procedure arguments online: https://riptutorial.com/tcl/topic/3365/procedure-arguments

Chapter 7: Regular Expressions

Syntax

- regexp ?switches? exp string ?matchVar? ?subMatchVar subMatchVar ...?
- regsub ?switches? exp string subSpec ?varName?

Remarks

This topic is not intended to discuss regular expressions themselves. There are many resources on the internet explaining regular expressions and tools to help build regular expressions.

This topic will try to cover the common switches and methods of using regular expressions in Tcl and some of the differences between Tcl and other regular expression engines.

Regular expressions are generally slow. The first question you should ask is "Do I really need a regular expression?". Only match what you want. If you don't need the other data, don't match it.

For the purposes of these regular expression examples, the -expanded switch will be used in order to be able to comment and explain the regular expression.

Examples

Matching

The `regexp` command is used to match a regular expression against a string.

```
# This is a very simplistic e-mail matcher.
# e-mail addresses are extremely complicated to match properly.
# there is no guarantee that this regex will properly match e-mail addresses.
set mydata "send mail to john.doe.the.23rd@no.such.domain.com please"
regexp -expanded {
    \y           # word boundary
    [^@\s]+      # characters that are not an @ or a space character
    @            # a single @ sign
    [\w.-]+      # normal characters and dots and dash
    \.           # a dot character
    \w+          # normal characters.
    \y           # word boundary
    } $mydata emailaddr
puts $emailaddr
john.doe.the.23rd@no.such.domain.com
```

The `regexp` command will return a 1 (true) value if a match was made or 0 (false) if not.

```
set mydata "hello wrld, this is Tcl"
# faster would be to use: [string match *world* $mydata]
if { [regexp {world} $mydata] } {
   puts "spelling correct"
```

```
} else {
   puts "typographical error"
}
```

To match all expressions in some data, use the -all switch and the -inline switch to return the data. Note that the default is to treat newlines like any other data.

```
# simplistic english ordinal word matcher.
set mydata {
    This is the first line.
    This is the second line.
    This is the third line.
    This is the fourth line.
    }
set mymatches [regexp -all -inline -expanded {
    \y                  # word boundary
    \w+                 # standard characters
    (?:st|nd|rd|th)     # ending in st, nd, rd or th
                        # The ?: operator is used here as we don't
                        # want to return the match specified inside
                        # the grouping () operator.
    \y                  # word boundary
    } $mydata]
puts $mymatches
first second third fourth
# if the ?: operator was not used, the data returned would be:
first st second nd third rd fourth th
```

Newline handling

```
# find real numbers at the end of a line (fake data).
set mydata {
    White 0.87 percent saturation.
    Specular reflection: 0.995
    Blue 0.56 percent saturation.
    Specular reflection: 0.421
    }
# the -line switch will enable newline matching.
# without -line, the $ would match the end of the data.
set mymatches [regexp -line -all -inline -expanded {
    \y                  # word boundary
    \d\.\d+             # a real number
    $                   # at the end of a line.
    } $mydata]
puts $mymatches
0.995 0.421
```

Unicode requires no special handling.

```
% set mydata {123ÂÃÄÈ456}
123ÂÃÄÈ456
% regexp {[[:alpha:]]+} $mydata match
1
% puts $match
ÂÃÄÈ
% regexp {\w+} $mydata match
1
```

```
% puts $match
123ÂÃÄÈ456
```

Documentation: regexp re_syntax

Mixing Greedy and Non-Greedy Quantifiers

If you have a greedy match as the first quantifier, the whole RE will be greedy,

If you have non-greedy match as the first quantifier, the whole RE will be non-greedy.

```
set mydata {
    Device widget1: port: 156 alias: input2
    Device widget2: alias: input1
    Device widget3: port: 238 alias: processor2
    Device widget4: alias: output2
    }
regexp {Device\s(\w+):\s(.*?)alias} $mydata alldata devname devdata
puts "$devname $devdata"
widget1 port: 156 alias: input2
regexp {Device\s(.*?):\s(.*?)alias} $mydata alldata devname devdata
puts "$devname $devdata"
widget1 port: 156
```

In the first case, the first \w+ is greedy, so all quantifiers are marked as greedy and the .*? matches more than is expected.

In the second case, the first .*? is non-greedy and all quantifiers are marked as non-greedy.

Other regular expression engines may not have an issue with greedy/non-greedy quantifiers, but they are much slower.

Henry Spencer wrote: *... The trouble is that it is very, very hard to write a generalization of those statements which covers mixed-greediness regular expressions -- a proper, implementation-independent definition of what mixed-greediness regular expressions should match -- and makes them do "what people expect". I've tried. I'm still trying. No luck so far. ...*

Substitution

The `regsub` command is used for regular expression matching and substitution.

```
set mydata {The yellow dog has the blues.}
# create a new string; only the first match is replaced.
set newdata [regsub {(yellow|blue)} $mydata green]
puts $newdata
The green dog has the blues.
# replace the data in the same string; all matches are replaced
regsub -all {(yellow|blue)} $mydata red mydata
puts $mydata
The red dog has the reds.
# another way to create a new string
regsub {(yellow|blue)} $mydata red mynewdata
puts $mynewdata
```

```
The red dog has the blues.
```

Using back-references to reference matched data.

```
set mydata {The yellow dog has the blues.}
regsub {(yellow)} $mydata {"\1"} mydata
puts $mydata
The "yellow" dog has the blues.
```

Documentation: regsub re_syntax

Differences Between Tcl's RE engine and other RE engines.

- \m : Beginning of a word.
- \M : End of a word.
- \y : Word boundary.
- \Y : a point that is not a word boundary.
- \Z : matches end of data.

Documentation: re_syntax

Matching a literal string with a regular expression

Sometimes you need to match a literal (sub-)string with a regular expression despite that substring containing RE metacharacters. While yes, it's possible to write code to insert appropriate backslashes to make that work (using `string map`) it is easiest to just prefix the pattern with `***=`, which makes the RE engine treat the rest of the string as just literal characters, disabling *all* further metacharacters.

```
set sampleText "This is some text with \[brackets\] in it."
set searchFor {[brackets]}

if {[ regexp ***=$searchFor $sampleText ]} {
    # This message will be printed
    puts "Found it!"
}
```

Note that this also means you can't use any of the anchors.

Read Regular Expressions online: https://riptutorial.com/tcl/topic/5205/regular-expressions

Chapter 8: Tcl Language Constructs

Syntax

- # This is a valid comment
- # This is a valid { comment }

Examples

Placing Comments

Comments in Tcl are best thought of as another command.
A comment consists of a `#` followed by any number of characters up to the next newline. A comment can appear wherever a command can be placed.

```
# this is a valid comment
proc hello { } {
  # the next comment needs the ; before it to indicate a new command is
  # being started.
  puts "hello world" ; # this is valid
  puts "dlrow olleh" # this is not a valid comment

  # the comment below appears in the middle of a string.
  # is is not valid.
  set hw {
      hello ; # this is not a valid comment
      world
      }

  gets stdin inputfromuser
  switch inputfromuser {
     # this is not a valid comment.
     # switch expects a word to be here.
     go {
       # this is valid.  The switch on 'go' contains a list of commands
       hello
     }
     stop {
       exit
     }
  }
}
```

Braces in comments

Due to the way the Tcl language parser works, braces in the code must be properly matched. This includes the braces in comments.

```
proc hw {} {
   # this { code will fail
   puts {hello world}
```

```
}
```

A *missing close-brace: possible unbalanced brace in comment* error will be thrown.

```
proc hw {} {
  # this { comment } has matching braces.
  puts {hello world}
}
```

This will work as the braces are paired up properly.

Quoting

In the Tcl language in many cases, no special quoting is needed.

These are valid strings:

```
abc123
4.56e10
my^variable-for.my%use
```

The Tcl language splits words on whitespace, so any literals or strings with whitespace should be quoted. There are two ways to quote strings. With braces and with quotation marks.

```
{hello world}
"hello world"
```

When quoting with braces, no substitutions are performed. Embedded braces may be escaped with a backslash, but note that the backslash is part of the string.

```
% puts {\{ \}}
\{ \}
% puts [string length {\{ \}}]
5
% puts {hello [world]}
hello [world]
% set alpha abc123
abc123
% puts {$alpha}
$alpha
```

When quoting with double quotes, command, backslash and variable substitutions are processed.

```
% puts "hello [world]"
invalid command name "world"
% proc world {} { return my-world }
% puts "hello [world]"
hello my-world
% puts "hello\tworld"
hello   world
% set alpha abc123
abc123
```

```
% puts "$alpha"
abc123
% puts "\{ \}"
{ }
```

Read Tcl Language Constructs online: https://riptutorial.com/tcl/topic/4470/tcl-language-constructs

Chapter 9: Variables

Syntax

- set ***varName ?value?***
- unset ?**-nocomplain**? ?--? ?*varName varName varName*?
- puts $*varName*
- puts [set *varName*]
- variable *varName*
- global *varName* ?***varName varName***?

Remarks

- Parameters enclosed within *?...?* such as *?varName?* represent optional arguments to a Tcl command.
- Documentation: global, upvar

Examples

Assigning values to variables

The command `set` is used to assign values in Tcl. When it is called with two arguments in the following manner,

```
% set tempVar "This is a string."
This is a string.
```

it places the second argument ("This is a string.") in the memory space referenced by the first argument (tempVar). `set` always returns the contents of the variable named in the first argument. In the above example, `set` would return "This is a string." without the quotes.

- If *value* is specified, then the contents of the variable *varName* are set equal to *value*.
- If *varName* consists only of alphanumeric characters, and no parentheses, it is a scalar variable.
- If *varName* has the form *varName(index)*, it is a member of an associative array.

Note that the name of the variable is not restricted to the Latin alphabet, it may consist of any combination of unicode characters (e.g. Armenian):

```
% set sոմն house
house
% puts ${sոմն}
house
```

Scoping

```
set alpha 1

proc myproc {} {
    puts $alpha
}

myproc
```

This code doesn't work because the two alphas are in different scopes.

The command `set alpha 1` creates a variable in the global scope (which makes it a global variable).

The command `puts $alpha` is executed in a scope that is created when the command `myproc` executes.

The two scopes are distinct. This means that when `puts $alpha` tries to look up the name `alpha`, it doesn't find any such variable.

We can fix that, however:

```
proc myproc {} {
    global alpha beta
    puts $alpha
}
```

In this case two global variables, `alpha` and `beta`, are linked to alias variables (with the same name) in the procedure's scope. Reading from the alias variables retrieves the value in the global variables, and writing to them changes the values in the globals.

More generally, the `upvar` command creates aliases to variables from any of the previous scopes. It can be used with the global scope (`#0`):

```
proc myproc {} {
    upvar #0 alpha alpha beta b
    puts $alpha
}
```

The aliases can be given the same name as the variable that is linked to(`alpha`) or another name (`beta` / `b`).

If we call `myproc` from the global scope, this variant also works:

```
proc myproc {} {
    upvar 1 alpha alpha beta b
    puts $alpha
}
```

The scope number `1` means "the previous scope" or "the caller's scope".

Unless you really know what you're doing, `#0`, `0`, and `1` are the only scopes that make sense to use with `upvar`. (`upvar 0` creates a local alias for a local variable, not strictly a scoping operation.)

Some other languages define scope by curly braces, and let code running in each scope see all names in surrounding scopes. In Tcl one single scope is created when a procedure is called, and only its own names are visible. If a procedure calls another procedure, its scope is stacked on top of the previous scope, and so on. This means that in contrast with C-style languages that only have global scope and local scope (with subscopes), each scope acts as an enclosing (though not immediately visible) scope to any scope it has opened. When a procedure returns, its scope is destroyed.

Documentation: global, upvar

Printing the value of a variable

In order to print the value of a variable such as,

```
set tempVar "This is a string."
```

The argument in the puts statement is preceded by a **$** sign, which tells Tcl to use the value of the variable.

```
% set tempVar "This is a string."
This is a string.
% puts $tempVar
This is a string.
```

Invoking set with one argument

`set` can also be invoked with just one argument. When called with just one argument, it returns the contents of that argument.

```
% set x 235
235
% set x
235
```

Deleting variable/s

The `unset` command is used to remove one or more variables.

```
unset ?-nocomplain? ?--? ?name name name name?
```

- Each *name* is a variable name specified in any of the ways acceptable to the `set` command.
- If a *name* refers to an element of an array then that element is removed without affecting the remainder of the array.
- If a *name* consists of an array name with no index in parentheses, then the entire array is deleted.
- If **-nocomplain** is given as the first argument, then all possible errors are suppressed from the command's output.
- The option -- indicates the end of the options, and should be used if you wish to remove a

variable with the same name as any of the options.

```
% set x 235
235
% set x
235
% unset x
% set x
can't read "x": no such variable
```

Namespace variables

The `variable` command ensures that a given namespace variable is created. Until a value is assigned to it, the variable's value is undefined:

```
namespace eval mynamespace {
    variable alpha
    set alpha 0
}
```

The variable can be accessed from outside the namespace (from anywhere, in fact) by attaching the name of the namespace to it:

```
set ::mynamespace::alpha
```

Access can be simplified within a procedure by using the `variable` command again:

```
proc ::mynamespace::myproc {} {
    variable alpha
    set alpha
}
```

This creates a local alias for the namespace variable.

For a procedure defined in another namespace, the variable name must contain the namespace in the invocation of `variable`:

```
proc myproc {} {
    variable ::mynamespace::alpha
    set alpha
}
```

Read Variables online: https://riptutorial.com/tcl/topic/3740/variables

Credits

S. No	Chapters	Contributors
1	Getting started with tcl	Brad Lanam, Community, Donal Fellows, klas2iop, Mallikarjunarao Kosuri, Peter Lewerin, stark, vasili111
2	Control Structures	Brad Lanam, Codename_DJ, Donal Fellows, Peter Lewerin, stark
3	Dictionaries	Brad Lanam, Donal Fellows, Peter Lewerin
4	Expressions	Donal Fellows, nurdglaw, Peter Lewerin, stark
5	Pathnames and filenames	Brad Lanam
6	Procedure arguments	Brad Lanam, Codename_DJ, Donal Fellows, klas2iop, Peter Lewerin
7	Regular Expressions	Brad Lanam, Donal Fellows
8	Tcl Language Constructs	Brad Lanam, stark
9	Variables	klas2iop, Peter Lewerin, stark